PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of: Sachin Mullick, et al.

Serial No. 10/668.467 Confirm 2942

Filed: 09/23/2003

For: Multi-Threaded Write Interface and Methods for Increasing the Single File Read and Write Throughput of a File Server

Group Art Unit: 2161

Examiner: Bibbee, Jared M.

Atty. Dkt. No.: 10830.0100.NPUS00

RULE 131 DECLARATION OF SORIN FAIBISH

Commissioner for Patents
PO Box 1450
Alexandria, VA 22313-1450

Sir:

1. I am a joint inventor of the subject patent application Ser. 10/668,467. I am presently employed by EMC Corporation, and I have been employed by EMC Corporation since at least 1997.

2. In the course of my work for EMC Corporation, together with Sachin Mullick, Jiannan Zheng, Xiaoye Jiang, and Peter Bixby, I was involved in the development of an invention related to a multi-threaded write interface, and the preparation of a patent application on this invention. I was responsible for reviewing and revising a first draft of the patent application in order to obtain a subsequent draft suitable for circulation among the other inventors.

3. Some time prior to Aug. 8, 2003, I received, via electronic mail, a first draft of a patent application on my multi-threaded write interface invention from my patent attorney, Richard C. Auchterlonie. Attached in Exhibit A is a true and correct copy of the transmittal E-mail message that I received with this first draft of the patent application, except that the date of the transmittal E-mail message has been redacted, where indicated by the box labeled "REDACTED".

4. On August 8, 2003, I completed a revision of the first draft of the patent application on my multi-threaded write interface invention, and I transmitted, via electronic mail, my revised version of the first draft of the patent application to my patent attorney, Richard C. Auchterlonie. Attached in Exhibit B is a true and correct copy of the transmittal E-mail and my revised version of the first draft of the patent application that was attached to this electronic mail on August 8, 2003, except that attorney-client communication has been redacted from the body of the E-mail, and attorney-client communication has been redacted from page 41 of my revised version of the first draft of the patent application, where indicated by boxes labeled "REDACTED".

5. On September 4, 2003, I received, via electronic mail, a second draft of a patent application on my multi-threaded write interface invention from my patent attorney, Richard C. Auchterlonie. Attached in Exhibit C is a true and correct copy of the transmittal E-mail that I received on September 4, 2003. I reviewed this second draft and circulated this second draft to Sachin Mullick, Jiannan Zheng, Xiaoye Jiang, and Peter Bixby, and we approved the filing of this second draft with the Patent and Trademark Office.

6. I hereby declare that all statements made of my own knowledge are true and that all statements made on information and belief are believed to be true, and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Respectfully submitted,

/ Faibish / 04/09/2009

Sorin Faibish date

Antoinette S Kelley

ANTOINETTE S. KELLEY
Notary Public
Commonwealth of Massachusetts
My Commission Expires
May 5, 2011

EXHIBIT A

Auchterlonie, Richard

From: Auchterlonie, Richard
Sent: REDACTED
To: sraidisn@emc.com
Cc: 'Clark_William@emc.com'; 'Mazzarella_Julie@emc.com'
Subject: EMC-03-062 A New Method for Enabling Simultaneous Parallel Writes to a Single File

Sorin:

Please find attached a first draft of the specification, drawings, declaration, and assignment for your patent application. Please review and pass it along to the other inventors as appropriate. If you would like any changes, please let me know and I will send you a revised draft. Otherwise, once all of the inventors have approved of the patent application, please contact Julie Mazzarella to arrange for execution of the patent application.

Thanks,

Richard C. Auchterlonie
Howrey Simon Arnold & White, LLP
750 Bering Drive
Houston, Texas 77057
Phone: (713) 787-1698
Fax: (713) 787-1440
AuchterlonieR@howrey.com


EMCR 100 PA Parallel Writes.DO...


EMCR 100 Drawings.pdf


EMCR100 Declaration.DOC


EMCR100 Assignment.DOC

EXHIBIT B

Auchterlonie, Richard

From: Sorin Faibish [sfaibish@emc.com]
Sent: Friday, August 08, 2003 4:35 PM
To: Auchterlonie, Richard
Cc: 'Clark_William@emc.com'; 'Mazzarella_Julie@emc.com'
Subject: Re: EMC-03-062 A New Method for Enabling Simultaneous Parallel Writes to a Single File

ParallelWritesPatentReview.zip...

Richard,

Attached please find the review of the above patent.

REDACTED

Thank you very much for your help.

"Auchterlonie, Richard" wrote:

> Sorin:
>
> Please find attached a first draft of the specification, drawings,
> declaration, and assignment for your patent application. Please review and
> pass it along to the other inventors as appropriate. If you would like any
> changes, please let me know and I will send you a revised draft. Otherwise,
> once all of the inventors have approved of the patent application, please
> contact Julie Mazzarella to arrange for execution of the patent application.
>
> Thanks,
>
> Richard C. Auchterlonie
> Howrey Simon Arnold & White, LLP
> 750 Bering Drive
> Houston, Texas 77057
> Phone: (713) 787-1698
> Fax: (713) 787-1440
> AuchterlonieR@howrey.com
>
> <<EMCR 100 PA Parallel Writes.DOC>> <<EMCR 100 Drawings.pdf>> <<EMCR100
> Declaration.DOC>> <<EMCR100 Assignment.DOC>>
>
> This email message and any files transmitted with it are subject to
> attorney-client privilege and contains confidential information intended
> only for the person(s) to whom this email message is addressed. If you have
> received this email message in error, please notify the sender immediately
> by telephone or email and destroy the original message without making a

PATENT

10830.0100.NPUS00

APPLICATION FOR UNITED STATES LETTERS PATENT

for

MULTI-THREADED WRITE INTERFACE AND METHODS FOR INCREASING THE SINGLE FILE READ AND WRITE THROUGHPUT OF A FILE SERVER

By

Sachin Mullick

Jiannan Zheng

Xiaoye Jiang

Sorin Faibish

Peter Bixby

Express Mail Mailing Label No. _____

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates generally to file servers and data processing networks. The present invention more particularly relates to a file server permitting concurrent writes from multiple clients to the same file. The invention specifically relates to increasing the single file write throughput of such a file server.

2. Description of the Related Art

Network data storage is most economically provided by an array of low-cost disk drives integrated with a large semiconductor cache memory. A number of data mover computers are used to interface the cached disk array to the network. The data mover computers perform file locking management and mapping of the network files to logical block addresses of storage in the cached disk array, and move data between network clients and the storage in the cached disk array.

Data consistency problems may arise if multiple clients or processes have concurrent access to read-write files. Typically write synchronization and file locking have been used to ensure data consistency. For example, the data write path for a file has been serialized by holding an exclusive lock on the file for the entire duration of creating a list of data buffers to be written to disk, allocating the actual on-disk storage, and writing to storage synchronously. Unfortunately, these methods involve considerable access delays due to contention for locks not only on the files but also on the file directories and a log used when committing data to storage. In order to reduce these delays, a file server may permit asynchronous writes in accordance with version 3 of the Network File System (NFS) protocol. See, for example, Vahalia et al. U.S. Patent

5,893,140 issued April 6, 1999, entitled "File Server Having a File System Cache and Protocol for Truly Safe Asynchronous Writes," incorporated herein by reference. More recently, byte range locking to a file has been proposed in version 4 of the NFS protocol. (See NFS Version 3 Protocol Specification, RFC 1813, Sun Microsystems, Inc., June 1995, incorporated herein by reference, and NFS Version 4 Protocol Specification, RFC 3530, Sun Microsystems, Inc., April 2003, incorporated herein by reference.)

Asynchronous writes and range locking alone will not eliminate access delays due to contention during allocation and commitment of file metadata. A Unix-based file in particular contains considerable metadata in the inode for the file and in indirect blocks of the file. The inode, for example, contains the date of creation, date of access, file name, and location of the data blocks used by the file in bitmap format. The NFS protocol specifies how this metadata must be managed. In order to comply with the NFS protocol, each time a write operation occurs, access to the file is not allowed until the metadata is updated on disk, both for read and write operations. In a network environment, multiple clients may issue simultaneous writes to the same large file such as a database, resulting in considerable access delay during allocation and commitment of file metadata.

SUMMARY OF THE INVENTION

In accordance with one aspect of the present invention, there is provided a method of operating a network file server for providing clients with concurrent write access to a file. The method includes the network file server responding to a concurrent write request from a client by obtaining a lock for the file, and then preallocating a metadata block for the file, and then releasing the lock for the file, and then asynchronously writing

to the file, and then obtaining the lock for the file, and then committing the metadata block to the file, and then releasing the lock for the file.

In accordance with another aspect, the invention provides a method of operating a network file server for providing clients with concurrent write access to a file. The method includes the network file server responding to a concurrent write request from a client by preallocating a block for the file, and then asynchronously writing to the file, and then committing the block to the file. The asynchronous writing to the file includes a partial write to a new block that has been copied at least in part from an original block of the file. The method further includes checking a partial block conflict queue for a conflict with a concurrent write to the new block, and upon finding an indication of a conflict with a concurrent write to the new block, waiting until resolution of the conflict with the concurrent write to the new block, and then performing the partial write to the new block.

In accordance with another aspect, the invention provides a method of operating a network file server for providing clients with concurrent write access to a file. The method includes the network file server responding to a concurrent write request from a client by preallocating a metadata block for the file, and then asynchronously writing to the file, and then committing the metadata block to the file. The method further includes gathering together preallocated metadata blocks for a plurality of client write requests to the file, and committing together the preallocated metadata blocks for the plurality of client write requests to the file by obtaining a lock for the file, committing the gathered preallocated metadata blocks for the plurality of client write requests to the file, and then releasing the lock for the file.

In accordance with yet another aspect, the invention provides a method of operating a network file server for providing clients with concurrent write access to a file. The method includes the network file server responding to a concurrent write request from a client by executing a write thread. Execution of the write thread includes obtaining an allocation mutex for the file, and then preallocating new metadata blocks that need to be allocated for writing to the file, and then releasing the allocation mutex for the file, and then issuing asynchronous write requests for writing to the file, waiting for callbacks indicating completion of the asynchronous write requests, obtaining the allocation mutex for the file, and then committing the preallocated metadata blocks, and then releasing the allocation mutex for the file.

In accordance with another aspect, the invention provides a network file server. The network file server includes storage for storing a file and at least one processor coupled to the storage for providing clients with concurrent write access to the file. The network file server is programmed for responding to a concurrent write request from a client by obtaining a lock for the file, and then preallocating a metadata block for the file, and then releasing the lock for the file, and then asynchronously writing to the file, and then obtaining the lock for the file, and then committing the metadata block to the file, and then releasing the lock for the file.

In accordance with another aspect, the invention provides a network file server. The network file server includes storage for storing a file, and at least one processor coupled to the storage for providing clients with concurrent write access to the file. The network file server is programmed for responding to a concurrent write request from a client by preallocating a block for the file, and then asynchronously writing to the file,

and then committing the block to the file. The network file server includes a partial block conflict queue for indicating a concurrent write to a new block that is being copied at least in part from an original block of the file. The network file server is programmed for responding to a client request for a partial write to the new block by checking the partial block conflict queue for a conflict, and upon finding an indication of a conflict, waiting until resolution of the conflict with the concurrent write to the new block of the file, and then performing the partial write to the new block of the file.

In accordance with another aspect, the invention provides a network file server. The network file server includes storage for storing a file, and at least one processor coupled to the storage for providing clients with concurrent write access to the file. The network file server is programmed for responding to a concurrent write request from a client by preallocating a metadata block for the file, and then asynchronously writing to the file; and then committing the metadata block to the file. The network file server is programmed for gathering together preallocated metadata blocks for a plurality of client write requests to the file, and committing together the preallocated metadata blocks for the plurality of client write requests to the file by obtaining a lock for the file, committing the gathered preallocated metadata blocks for the plurality of client write requests to the file, and then releasing the lock for the file.

In accordance with yet still another aspect, the invention provides a network file server. The network file server includes storage for storing a file, and at least one processor coupled to the storage for providing clients with concurrent write access to the file. The network file server is programmed with a write thread for responding to a concurrent write request from a client by obtaining an allocation mutex for the file, and

then preallocating new metadata blocks that need to be allocated for writing to the file, and then releasing the allocation mutex for the file, and then issuing asynchronous write requests for writing to the file, waiting for callbacks indicating completion of the asynchronous write requests, and then obtaining the allocation mutex for the file, and then committing the preallocated metadata blocks, and then releasing the allocation mutex for the file.

In accordance with a final aspect, the invention provides a network file server. The network file server includes storage for storing a file, and at least one processor coupled to the storage for providing clients with concurrent write access to the file. The network file server is programmed for responding to a concurrent write request from a client by preallocating a block for writing to the file, asynchronously writing to the file, and then committing the preallocated block. The network file server also includes an uncached write interface, a file system cache, and a cached read-write interface. The uncached write interface bypasses the file system cache for sector-aligned write operations, and the network file server is programmed to invalidate cache blocks in the file system cache including sectors being written to by the cached read-write interface.

BRIEF DESCRIPTION OF THE DRAWINGS

Other objects and advantages of the invention will become apparent upon reading the following detailed description with reference to the accompanying drawings wherein:

FIG. 1 is a block diagram of a data processing system including multiple clients and a network file server;

FIG. 2 is a block diagram showing further details of the network file server in the data processing system of FIG. 1;

FIG. 3 is a block diagram of various read and write interfaces in a Unix-based file system layer (UxFS) in the network file server of FIG. 2;

FIG. 4 shows various file system data structures associated with a file in the network file server of FIG. 2;

FIGS. 5 and 6 comprise a flowchart of programming in the Common File System (CFS) layer in the network file server for handling a write request from a client;

FIG. 50 comprise a diagram showing multiple read or write I/Os pipelined into parallel streams in the Common File System (CFS) layer in the network file server for handling concurrent read and write requests from a client;

FIG. 5a comprise a flowchart of programming in the Common File System (CFS) layer in the network file server for handling a read request from a client;

FIG. 5b comprise a flowchart of programming in the Common File System (CFS) layer in the network file server for handling concurrent read and write requests from a client;

FIG. 7 is a flowchart of a write thread in the UxFS layer of the network file server;

FIG. 8 is a more detailed flowchart of steps in the write thread for committing preallocated metadata;

FIG. 9 is a block diagram of a partial block write during a copy-on-write operation;

FIG. 10 is a block diagram of a read-write file as maintained by the UxFS layer;

FIG. 11 is a block diagram of the read-write file of FIG. 10 after creation of a read-only version of read-write file;

FIG. 12 is a block diagram of the read-write file of FIG. 11 after a copy-on-write operation upon a direct block and two indirect blocks between the direct block and the inode of the read-write file;

FIG. 13 is a flowchart of steps in a write thread for performing the partial block write operation of FIG. 9; and

FIG. 14 shows a flowchart of steps in a write thread for allocating file blocks when writing to a file having read-only versions.

While the invention is susceptible to various modifications and alternative forms, specific embodiments thereof have been shown in the drawings and will be described in detail. It should be understood, however, that it is not intended to limit the invention to the particular forms shown, but on the contrary, the intention is to cover all modifications, equivalents, and alternatives falling within the scope of the invention as defined by the appended claims.

DESCRIPTION OF ILLUSTRATIVE EMBODIMENTS

FIG. 1 shows an Internet Protocol (IP) network 20 including a network file server 21 and multiple clients 23, 24, 25. The network file server 21, for example, has multiple data mover computers 26, 27, 28 for moving data between the IP network 20 and a cached disk array 29. The network file server 21 also has a control station 30 connected via a dedicated dual-redundant data link 31 among the data movers for configuring the data movers and the cached disk array 29.

Further details regarding the network file server 21 are found in Vahalia et al., U.S. Patent 5,893,140, incorporated herein by reference, and Xu et al., U.S. Patent 6,324,581, issued Nov. 27, 2001, incorporated herein by reference. The network file server 21 is managed as a dedicated network appliance, integrated with popular network operating systems in a way, which, other than its superior performance, is transparent to the end user. The clustering of the data movers 26, 27, 28 as a front end to the cached disk array 29 provides parallelism and scalability. Each of the data movers 26, 27, 28 is a high-end commodity computer, providing the highest performance appropriate for a data mover at the lowest cost. The data mover computers 26, 27, 28 may communicate with the other network devices using standard file access protocols such as the Network File System (NFS) or the Common Internet File System (CIFS) protocols, but the data mover computers do not necessarily employ standard operating systems. For example, the network file server 21 is programmed with a Unix-based file system that has been adapted for rapid file access and streaming of data between the cached disk array 29 and the data network 20 by any one of the data mover computers 26, 27, 28.

FIG. 2 shows software modules in the data mover 26 introduced in FIG. 1. The data mover 26 has a Network File System (NFS) module 41 for supporting communication among the clients and data movers of FIG. 1 over the IP network 20 using the NFS file access protocol, and a Common Internet File System (CIFS) module 42 for supporting communication over the IP network using the CIFS file access protocol. The NFS module 41 and the CIFS module 42 are layered over a Common File System (CFS) module 43, and the CFS module is layered over a Universal File System

(UxFS) module 44. The UxFS module supports a UNIX-based file system, and the CFS module 43 provides higher-level functions common to NFS and CIFS.

The UxFS module accesses data organized into logical volumes defined by a module 45. Each logical volume maps to contiguous logical storage addresses in the cached disk array 29. The module 45 is layered over a SCSI driver 46 and a Fibre-channel protocol (FCP) driver 47. The data mover 26 sends storage access requests through a ~~network interface card~~host bus adapter 48 using the SCSI protocol or the Fibre-Channel protocol, depending on the physical link between the data mover 26 and the cached disk array 29.

A network interface card 49 in the data mover 26 receives IP data packets from the IP network 20. A TCP/IP module 50 decodes data from the IP data packets for the TCP connection and stores the data in message buffers 53. For example, the UxFS layer 44 writes data from the message buffers 53 to a file system 54 in the cached disk array 29. The UxFS layer 44 also reads data from the file system 54 or a file system cache 51 and copies the data into the message buffers 53 for transmission to the network clients 23, 24, 25.

To maintain the file system 54 in a consistent state during concurrent writes to a file, the UxFS layer maintains file system data structures 52 in random access memory of the data mover 26. To enable recovery of the file system 54 to a consistent state after a system crash, the UxFS layer writes file metadata to a log 55 in the cached disk array during the commit of certain write operations to the file system 54.

FIG. 3 shows various read and write interfaces in the UxFS layer. These interfaces include a cached read/write interface 61 for accessing the file system cache 51, an uncached multi-threaded write interface 63, and an uncached read interface 64.

The cached read/write interface 61 permits reads and writes to the file system cache 51. If data to be accessed does not reside in the cache, it is staged from the file system 54 to the file system cache 51. Data written to the file system cache 51 from the cached read/write interface 61 is written down to the file system cache during a commit operation. The file data is written down first, followed by writing of new file metadata to the log 55 and then writing of the new metadata to the file system 54.

The uncached multi-threaded write interface 63 is used for sector-aligned writes to the file system 54. Sectors of data (e.g., 512 byte blocks) are read from the message buffers (53 in FIG. 2) and written directly to the cached disk array 29. For example, each file block is sector aligned and is 8 K bytes in length. When a sector-aligned write occurs, any cache blocks in the file system cache that include the sectors being written to are invalidated. In effect, the uncached multi-threaded write interface 63 commits file data when writing the file data to the file system 54 in storage. The uncached multi-threaded write interface 63 allows multiple concurrent writes to the same file. If a sector-aligned write changes metadata of a file such as file block allocations, then after the data of the file has been written, the new metadata is written to the log 55, and then the new metadata is written to the file system 54. The new metadata includes modifications to the file's inode, any new or modified indirect blocks, and any modified quota reservation.

The uncached read interface 64 reads sectors of data directly from the file system 54 into the message buffers (53 in FIG. 2). For example, the read request must have a

sector aligned offset and specifies a sector count for the amount of data to be read. The data can be read into multiple message buffers in one input/output operation so long as the sectors to be read are in contiguous file system blocks.

Typically, the cached read/write interface 61 is used for reading data from read-write files and from any read-only versions of the read-write files. The uncached write interface 63 is used for sector-aligned writes to read-write files. If the writes are not sector aligned, then the cached read-write interface 61 is used. The uncached read interface 64 is used for sector-aligned reads when there is no advantage to retaining the data in the file system cache 51; for example, when streaming data to a remote copy of a file.

FIG. 4 shows various file system data structures 52 associated with a file. A virtual node (VNODE) 71 represents the file. The virtual node 71 is linked to an allocation mutex (mutually exclusive lock) 72, a partial block conflict queue 73, a partial write wait queue 74, an input-output (I/O) list 75, a staging queue 76, and preallocation block lists 77. When a file block is preallocated, it is reserved for use in the on-disk file system 54. A preallocated file block can be linked into the in-memory file block structure in the file system cache 51 as maintained by the UxFS layer 44, and later the preallocated file block can become part of the on-disk file system 54 when the preallocated file block is committed to storage. (An example of the file block structure is shown in FIG. 10.) The write threads of the uncached multi-threaded write interface (63 in FIG. 3) use the allocation mutex 72 for serializing preallocation of file metadata blocks and commitment of the preallocated metadata blocks. For a Unix-based file, the preallocated metadata blocks include new indirect blocks, which are added to the file

when the file is extended. As described below with reference to FIGS. 11 to 12, one or more new indirect blocks may also be added to a read-write file system when processing a client request to write to a direct block that is shared between the read-write file system and a read-only version of the read-write file system.

Preallocation of the file metadata blocks under control of the allocation mutex prevents multiple writers from allocating the same metadata block. The actual data write is done using asynchronous callbacks within the context of the thread, and does not hold any locks. Since writing to the on-disk storage takes the majority of the time, the preallocation method enhances concurrency, while maintaining data integrity.

The preallocation method allows concurrent writes to indirect blocks within the same file. Multiple writers can write to the same indirect block tree concurrently without improper replication of the indirect blocks. Two different indirect blocks will not be allocated for replicating the same indirect block. The write threads use the partial block conflict queue 73 and the partial write wait queue 74 to avoid conflict during partial block write operations, as further described below with reference to FIG. 9.

The I/O list 75 maps the message buffers (53 in FIG. 2) to data blocks to be written. The write threads use the I/O list 75 to implement byte range locking. The data blocks, for example, are 512 bytes in length providing sector-level granularity for the byte range locking. Alternatively, the data block length is a multiple of the sector size.

In order to prevent the log (55 in FIG. 2) from becoming a bottleneck, the preallocated metadata blocks for multiple write threads writing to the file at the same time are committed together under the same logging lock. Committing more than one allocation under one lock increases the throughput. For this purpose, a staging queue 76

is allocated and linked to the file virtual node 71. Preallocation block lists 77 identify the respective preallocated metadata blocks for the write threads writing to the file. The staging queue 76 receives pointers to the preallocation block lists 77 of the write threads waiting for the allocation mutex 72 of the file for commitment of their preallocated metadata blocks. For example, the staging queue 76 is a conventional circular queue, or the preallocation block lists 77 are linked together into a circular list to form the staging queue. There can be multiple files, and each file can have a respective staging queue waiting for commitment of the file's preallocation block lists. A wait list of staging queues 78 identifies the staging queues waiting for service on a first-come, first-served basis.

From a client's view, the write operation performed by a write thread in the uncached write interface is a synchronous operation. The write thread does not return an acknowledgement to the client until the write data has been written down to the file system in storage, and the metadata allocation has been committed to storage.

FIGS. 5 and 6 show programming in the Common File System (CFS) layer in the network file server for handling a write request from a client. In a first step 81, if the uncached multi-threaded write interface (63 in FIG. 3) is not turned on for the file system, then execution branches to step 82. For example, the uncached interface can be turned on or off per file system as a mount-time option. In step 82, the CFS layer obtains an exclusive lock upon the file, for example by acquiring the allocation mutex (72 in FIG. 4) for the file. Then in step 83, the CFS layer writes a specified number of bytes from the source to the file, starting at a specified byte offset, using the cached read/write interface (61 in FIG. 3). The source, for example, is one or more of the message buffers (53).

Then in step 84, the CFS layer releases the exclusive lock upon the file, and processing of the write request is finished.

In step 81, if the uncached multi-threaded write interface is turned on for the file system, then execution continues to step 85. In step 85, if the write data specified by the write request is not sector aligned (or the data size is not in multiple sectors), then execution branches to step 82. Otherwise, execution continues from step 85 to step 86.

In step 86, the CFS layer acquires a shared lock upon the file. The shared lock prevents the CFS layer from obtaining an exclusive lock upon the file for a concurrent write request (e.g., in step 82). However, as described below, the shared lock upon the file does not prohibit write threads in the UxFS layer from acquiring the allocation mutex (72 in FIG. 4) during the preallocation of metadata blocks or during the commitment of the metadata blocks.

In step 87, the CFS layer checks the I/O list (75 in FIG. 4) for a conflict. If there is a conflicting data block on the I/O list, then execution waits until the conflicting data block is flushed out of the I/O list. In certain clustered systems in which direct data access to the file in the data storage is shared with other servers or clients, execution may also wait in step 87 for range locks to be released by another server or client sharing direct access to the file. After step 87, execution continues to step 88 in FIG. 6.

In step 88 of FIG. 6, the CFS layer writes the specified number of bytes from the source to the file, starting at a specified sector offset, using the uncached multi-threaded write interface (63 in FIG. 3). Then in step 89, the CFS layer invalidates any cached entries for the file system blocks that have been written to in the file system cache (51 in FIG. 3). The invalidation occurs after completion of any reads in progress to these file

system blocks. In step 90, the CFS layer releases the shared lock upon the file, and processing of the write request is finished.

The parallel write architecture can be used to achieve pipelining, since the data write stage does not involve any metadata interactions. Figure 50 shows how write pipelining is achieved by using the preallocation, write, and commit architecture. The write is divided into three steps, namely preallocation, write, and commit. The preallocation stage is achieved synchronously, and an allocation mutex prevents multiple preallocations from occurring simultaneously for the same file. Once the metadata preallocation stage is complete, the data transfer stage can be independently handled by another virtual processing unit, using Hyper threading technology (Jackson technology) [http://developer.intel.com/design/Pentium4/manuals/IntelR_PentiumR_4_Processor_-_Manuals.url]. The data transfer request can be handed over to another virtual processing unit, and it does not need any interaction with the original processor that does the metadata management. Separate processing units can service data read and write requests generated by the master processor that handles metadata management. The write list can be handed over to a separate processing unit that will then go through the write request, take the data from the network packets, write it to the disk locations specified by the data write request created by the master processor, and complete the data write to the disk from the network packets.

The actual data writes happen concurrently, and can be handled by different engines or virtual processing units. The final commit of the allocations are gathered together and committed under the same allocation mutex. The data writes to disk are the longest stage. With pipelining, the writes can be achieved continuously. This results in

an increase in the number of write operations that can be performed in a given time period. The architecture allows the next write metadata preallocation to occur while other engines are processing the data write to disk.

When write I/O requests arrive at the master processor or thread the request is analyzed and if there are any metadata operations associated, S1 in Fig. 50, they are executed by the master processor while the block write I/O is pipelined to another separate processing unit. The processing unit will pipeline multiple write I/Os, S2 in Fig 50, and will commit all the write I/Os to the disk independently of the metadata operation. At the end of the data commit process the metadata will be committed, S3 in Fig 50, to the disk as well. It must be noted that the master processor is freed to perform additional metadata management operations while the virtual processing unit write the I/O to the disk. There could be a pool of virtual processing units that execute the write tasks and they can be allocated for additional processing tasks by the master processor. Some tasks are executed only by the master processor. The master processor is preallocated when the data mover is rebooted. All the processing of the pipeline is based on the fact that the writes are uncached and there is no contingency or locking to the files. If there are any contingencies they are solved by the master processor before the writes are pipelined.

FIG. 5a shows a flowchart of programming in the CFS layer in the network file server for handling a read request from a client simultaneous with handling a write request to the same file. [Please describe it in your words].

Fig. 5b presents the behavior of the server when there are Read – Write interactions during concurrent access of multiple I/O threads to a single file.

More specific this describes the case when read I/O requests are sent for blocks to which there are concurrent ongoing writes. There are 2 processes that are described: the modifications in the write I/O flow and the read I/O flow described in the next paragraphs. Fig. 5a describe the path of the reads to a file that is written to, while Fig. 5b describe the concurrent writes and reads.

- Write request

  Find Write Request Range -> Find partial blocks and create a partial block list -> Preallocate the metadata blocks for the range of block numbers in the inode that is being written -> Send the aynchronous write requests -> Wait for asynchronous write request ->Get blockCommit Lock -> Commit the preallocated metadata blocks for the range we have written to the inode -> release blockCommitLock -> start asynchronous writes for conflict I/Os -> Find range of blocks in the buffer cache to be clobbered -> Clobber the buffer Cache for the block range being commited -> if active readers, mark the cache range as clobberStaleData.

- Read request

  Find Read Request Range -> Is Data in Cache – Yes -> Read data from Cache -> End

  --No-> get blockCommit Lock -> Get the commited mapping from the inode for the read request range -> release blockCommitLock -> read data from disk to the buffer cache and source -> If clobberStaleData flag set in this block range, clobber the buffer cache, as some write was done while we were reading the data and we do not want to cache the data.

The above processes are based on the following assumptions:

– Writes shouldn't be blocked

- Reads can read old data, new data, or a mix
- This doesn't address overlapping writes (for now).

Reads will work as implemented for normal file access non-concurrent with writes. The *bufmap* will be examined. Valid hints will result in references on the buffer, missing blocks will be marked as IOP (IO in Progress) and the generation count will be set to a value associated with this read, and then a read will be started. After completing any reads necessary the blocks that were previously marked as IOP will be cleared in one of the following ways:

- If the slot is cleared, then it's been purged and the just completed read should not be entered and cached.
- If it's marked as IOP then the generation count is checked:
    - If the generation count is the same as we set then we cache our hint,
    - otherwise we ignore this entry we can use the data to satisfy the read.

Writes will simply be allowed to proceed. At the end of the write we'll go back in do an invalidation of the entire range of blocks written. If the slot was empty it's ignore, if the slot had a hint it's cleared, and if the slot was IOP the IOP will be cleared and any waiters will be awoken.

While the above only works with the assumptions listed, I believe it should be possible to use something similar to satisfy the following assumptions:

- Writes shouldn't be blocked
- Reads can read old data or new data, but not a mix
- Overlapping writes must be controlled.

To change the processing to fulfill this set of requirements we'd first need to prevent the mixed read case. This could be done by either completing the read completely or releasing the buffers and restarting the reads when we see a conflict (case 2b above).

Further write overlaps could be prevented by adding a new type of entry in the *bufmap* which would indicate a "Write In Progress" (WIP). Each write would enter WIP in each slot in the range flushing any old entries. Subsequent writes that encountered a WIP bit within it's own range would be required to wait. Likewise attempts to read that encounter a WIP would need to block.

FIG. 7 shows a flowchart of a write thread in the UxFS layer (44 in FIG. 2). In a first step 101, the write thread gets the allocation mutex (72 in FIG. 4) for the file. Then in step 102, the write thread preallocates metadata blocks for the block range being written to the file. In step 103, the write thread releases the allocation mutex for the file.

In step 104, the write thread issues asynchronous write requests for writing to blocks of the file. For example, a list of callbacks is created. There is one callback for each asynchronous write request consisting of up to 64 K bytes of data from one or more contiguous file system blocks. An I/O list is created for each callback. The asynchronous write requests are issued asynchronously, so multiple asynchronous writes may be in progress concurrently. In step 105, the write thread waits for the asynchronous write requests to complete.

In step 106, the write thread gets the allocation mutex for the file. In step 107, the write thread commits the preallocated metadata blocks to the file system in storage. The new metadata for the file including the preallocated metadata blocks is committed by

being written to the log (55 in FIG. 3). File system metadata such as the file modification time, however, is not committed in step 107 and is not logged. Instead, file system metadata such as the file modification time is updated at a file system sync time during the flushing of file system inodes. Finally, in step 108, the write thread releases the allocation mutex for the file. This method of preallocating and committing metadata blocks does not need any locking or metadata transactions for re-writing to allocated blocks.

FIG. 8 is a more detailed flowchart of steps in the write thread for committing the preallocated metadata. In a first step 111, if there is not a previous commit in progress, then execution continues to step 112. In step 112, the thread gets the allocation mutex for the file. Then in step 113, the thread writes new metadata (identified by the thread's preallocation list) to the log in storage. In step 114, the thread writes the new metadata (identified by the thread's preallocation list) to the file system in storage. In step 115, the thread releases the allocation mutex for the file. Finally, in step 116, the thread returns an acknowledgement of the write operation.

In step 111, if there was a previous commit in progress, then the thread inserts a pointer to the threads' preallocation list onto the tail of the staging queue for the file. If the staging queue was empty, then the staging queue is put on the wait list of staging queues (78 in FIG. 4). The thread is suspended, waiting for a callback from servicing of the staging queue. In step 118, the metadata identified by the thread's preallocation list is committed when the staging queue is serviced. The staging queue is serviced by obtaining the allocation mutex for the file, writing the new metadata for all of the preallocation lists on the staging queue to the log in storage, then writing this new

metadata to the file system in storage, and then releasing the allocation mutex for the file. Once servicing of the staging queue has committed the new metadata for the thread's preallocation list, execution of the thread is resumed in step 116 to return an acknowledgement of the write operation. After step 116, the thread is finished with the write operation.

FIG. 9 is a block diagram of a partial block write during a copy-on-write operation. Such an operation involves copying a portion of the data from an original file system block 121 to a newly allocated file system block 123, and writing a new partial block of data 122 to the newly allocated file system block. The portion of the data from the original file system block becomes merged with the new partial block of data 122. If the new partial block of data is sector aligned, then the partial block write can be performed by the uncached multi-threaded write interface (63 in FIG. 3). Otherwise, if the new partial block of data were not sector aligned, then the partial block write would be performed by the cached read/write interface (61 in FIG. 3).

The copy-on-write operation may frequently occur in a file system including one or more read-only file versions of a read-write file. Such a file system is described in Chutani, Sailesh, et al., "The Episode File System," Carnegie Mellon University IT Center, Pittsburgh, PA, June 1991, incorporated herein by reference. Each read-only file version is a snapshot of the read-write file at a respective point in time. Read-only file versions can be used for on-line data backup and data mining tasks.

In a copy-on-write file versioning method, the read-only file version initially includes only a copy of the inode of the original file. Therefore the read-only file version initially shares all of the data blocks as well as any indirect blocks of the original file.

When the original file is modified, new blocks are allocated and linked to the original file inode to save the new data, and the original data blocks are retained and linked to the inode of the read-only file version. The result is that disk space is saved by only saving the difference between two consecutive versions. This process is shown in FIGS. 9, 10, and 11.

FIG. 10 shows a read-write file as maintained by the UxFS layer. The file has a hierarchical organization, depicted as an inverted tree. The file includes a read-write inode 131, a direct block 132 and an indirect block 133 linked to the read-write inode, a direct block 134 and an indirect block 135 linked to the indirect block 133, and direct blocks 136 and 137 linked to the indirect block 135.

When a read-only version of a read-write file is created, a new inode for the read-only version is allocated. The read-write file inode and file handle remain the same. After allocation of the new inode, the read-write file is locked and the new inode is populated from the contents of the read-write file inode. Then the read-write file inode itself is modified, the transaction is committed, and the lock on the read-write file is released.

The allocation of blocks during the copy-on-write to the read-write file raises the possibility of the supply of free storage being used up after writing to a small fraction of the blocks of the read-write file. To eliminate this possibility, the read-write file can be provided with a "persistent reservation" mechanism so that the creation of a read-only version will fail unless there can be reserved a number of free storage blocks equal to the number of blocks that become shared between the read-only version and the read-write file. The number of reserved blocks can be maintained as an attribute of the file. The

number of reserved blocks for a read-only file can be incremented as blocks become shared with a read-only version, and decremented as blocks are allocated during the writes to the read-write file.

FIG. 11 shows the read-write file of FIG. 10 after creation of a read-only version of the read-write file. The read-only inode 138 is a copy of the read-write inode 131. The read-write inode 131 has been modified to indicate that the direct block 132 and the indirect block 133 are shared with a read-only version. For example, in the read-write inode 131, the most significant bit in each of the pointers to direct block 132 and the indirect block 133 have been set to indicate that the pointers point to blocks that are shared with the read-write file. (The links represented by such pointers to shared blocks are indicated by dotted lines in FIGS. 11 and 12.) Also, by inheritance, any and all of the descendants of a shared block are also shared blocks. Routines in the UxFS layer that use the pointers to locate the pointed-to file system blocks simply mask out the most significant to determine the block addresses.

In general, for the case in which there are multiple versions of a file sharing file blocks, when a file block is shared, it is desirable to designate the oldest version sharing the block to be the owner of the block, and any other files to be non-owners of the block. A pointer in a non-shared block pointing to a shared block will have its most significant bit set if the block is not owned by the owner of the non-shared block, and will have its most significant bit clear if the block is owned by the owner of the non-shared block.

When writing to a specified sector of a file, a search of the file block hierarchy is done starting with the read-write inode, in order to find the file block containing the specified sector. Upon finding a pointer indicating that the pointed-to block is shared, the

pointed-to block and its descendants are noted as "copy on write" blocks. If the specified sector is found in a "copy on write" block, then a new file block is allocated.

In practice, multiple write threads are executed concurrently, so that more than one concurrent write thread could determine a need to preallocate the same new file block. The allocation mutex is used to serialize the allocation process so more than one preallocation of a new file block does not occur. For example, once the write thread has obtained the allocation mutex, the write thread then determines whether a new block is needed, and if so, then the write thread preallocates the new block. The write thread may obtain the allocation mutex, allocate multiple new blocks in this fashion, and then release the allocation mutex. For example, to write to a direct block of a file, when the write thread finds a shared block on the path in the file hierarchy down to the direct block of the file, the write thread obtains the allocation mutex, and then allocates all the shared blocks that it then finds down the path in the file hierarchy down to and including the direct block, and then release the allocation mutex.

Once a new file block has been allocated, a partial block write to the new file block is performed, unless the write operation writes new data to the entire block. The new file block is the same type (direct or indirect) as the original "copy on write" file block containing the specified sector. If the write operation writes new data to the entire new file block, then no copy need be done and the new data is simply written into the newly allocated block. (A partial write could be performed when the write operation writes new data to the entire block, although this would not provide the best performance.)

If the read-write inode or a block owned by the read-write file was a parent of the original "copy on write" block, then the new file block becomes a child of the read-write inode or the block owned by the read-write file. Otherwise, the new file block becomes the child of a newly allocated indirect block. In particular, copies are made of all of the "copy on write" indirect blocks that are descendants of the read-write inode and are also predecessors of the original "copy on write" file block.

For example, assume that a write request specifies a sector found to be in the direct block 137 of FIG. 11. Upon searching down the hierarchy from the read-write inode 131, it is noted that indirect blocks 133 and 135 and the direct block 137 are "copy on write" blocks. As shown in FIG. 12, new indirect blocks 139 and 140 and a new direct block 141 have been allocated. The new direct block 141 is a copy of the original direct block 136 except that it includes the new data of the write operation. The new indirect block 140 is a copy of the original indirect block 135 except it has a new pointer pointing to the new direct block 141 instead of the original direct block 137. The new indirect block 139 is a copy of the original indirect block 133 except it has a new pointer pointing to the new indirect block 140 instead of the original indirect block 135. Also, the read-write inode 131 has been modified to replace the pointer to the original indirect block 133 with a pointer to the new indirect block 139.

In some instances, a write to the read-write file will require the allocation of a new direct block without any copying from an original direct block. This occurs when there is a full block write, a partial block write to a hole in the file, or a partial block write to an extended portion of a file. When there is a partial block write to a hole in the file or a partial block write to the extended portion of a file, the partial block of new data is

written to the newly allocated direct block, and the remaining portion of the newly allocated direct block is filled in with zero data.

It is possible that the UxFS layer will receive multiple concurrent writes that all require new data to be written to the same newly allocated block. These multiple concurrent writes need to be synchronized so that only one new block will be allocated and the later one of the threads will not read old data from the original block and copy the old data onto the new data from an earlier one of the threads. The UxFS layer detects the first such write request and puts a corresponding entry into the partial block conflict queue (73 in FIG. 4). The UxFS layer detects the second such write request, determines that it is conflicting upon inspection of the partial block conflict queue, places an entry to the second such write request in the partial write wait queue (74 in FIG. 4), and suspends the write thread for the second such write request until the conflict is resolved.

FIG. 13 is a flowchart of steps in a write thread for performing the partial block write operation of FIG. 9. In a first step 151 of FIG. 13, if the newly allocated file system block (124 in FIG. 9) is not on the partial block conflict queue (73 in FIG. 4), then execution branches to step 152. In step 152, the partial block write thread puts the new block on the partial block conflict queue. In step 153, the partial block write thread copies data that will not be overwritten by the partial block write, the data being copied from the original file system block to the new file system block. In step 154, asynchronous write operations are performed to write the new partial block of data to the new block. In step 155, the partial block write thread gets the allocation mutex for the file, commits the preallocated metadata (or the preallocated metadata is gathered and committed upon servicing of the staging queue if a previous commit is in progress),

removes the new block from the partial block conflict queue, issues asynchronous writes for any corresponding blocks on the partial write wait queue, and releases the allocation mutex.

In step 151, if the newly allocated file system block was on the partial block conflict queue, then execution continues to step 156. In step 156, the partial block write thread puts a write callback on the partial write wait queue for the file. Then execution is suspended until the callback occurs (from the completion of the asynchronous writes issued in step 155). Upon resuming, in step 157, the partial block write thread gets the allocation mutex for the file, commits the preallocated metadata (or the preallocated metadata is gathered and committed upon servicing of the staging queue if a previous commit is in progress), and releases the allocation mutex.

FIG. 14 shows steps in a write thread for allocating file blocks when writing to a file having read-only versions. In a first step 161, if the file block being written to is not shared with a read-only version, then execution branches to step 162 to write directly to the block without any transaction. In other words, there is no need for allocating any additional blocks.

In step 161, if the file block being written to is shared with a read-only version, then execution continues to step 163. In step 163, if the file block being written to is an indirect block, then execution branches to step 164. In step 164, a new indirect block is allocated, the original indirect block content is copied to the new indirect block, and the new metadata is written to the new indirect block synchronously. If the block's parent is an indirect block shared with a read-only version, then a new indirect block is allocated for copy-on-write of the new block pointer. Any other valid block pointers in this new

indirect block point to shared blocks, and therefore the most significant bit in each of these other valid block pointers should be set (as indicated by the dotted line between the indirect blocks 136 and 140 in FIG. 12). For example, just after the original indirect block content is copied to the new indirect block, the most significant bit is set in all valid block pointers in the new indirect block. As described above with respect to FIG. 12, this copy-on-write may require one or more additional indirect blocks to be allocated (such as indirect block 139 in FIG. 12). For example, the tree of a UxFS file may include up to three levels of indirect blocks. All of the file blocks that need to be allocated can be predetermined so that the allocation mutex for the file can be obtained, all of the new blocks that are needed can be allocated together, and then the allocation mutex for the file can be released.

In step 163, if the file block being written to is not an indirect block, then execution continues to step 165. This is the case in which the file block being written to is a direct block. In step 165, if the write to the file block is not a partial write, then execution branches to step 166. In step 166, a new direct block is allocated and the block of new data is written directly to the new direct block. If the original block's parent is an indirect block that is shared with a read-only version, then a new indirect block is allocated for copy-on-write of the new block pointer. As described above with respect to FIG. 12, this copy-on-write may require one or more additional indirect blocks to be allocated.

In step 167, for the case of a partial write, execution continues from step 156 to step 167 to use the partial write technique as described above with respect to FIG. 9 and FIG. 13.

Various parts of the programming for handling a write thread the UxFS layer have been described above with reference to FIGS. 7 to 14. Following is a listing of the steps in the preferred implementation of this programming.

1. The write thread receives a write request specifying the source and destination of the data to be written. The source is specified in terms of message buffers and the message buffer header size. The destination is specified in terms of an offset and number of bytes to be written.

2. The write thread calculates the starting and ending logical block number, total block count, and determines whether the starting and ending blocks are partial blocks.

3. The write thread gets the allocation mutex for the file.

4. The write thread searches the file tree along a path from the file inode to the destination file blocks to determine whether there are any shared blocks along this path. For each such shared block, a new direct or indirect block is allocated synchronously, as described above with reference to FIGS. 11, 12, and 14.

5. The write thread identifies partial blocks of write data using the starting physical block number and the number of blocks to be written. Only the starting and ending block to be written can be partial. Also, if some other thread got to these blocks first, the block mapping may already exist and the "copy-on-write" will be done by the prior thread. The partial block conflict queue is checked to determine whether such an allocation and "copy-on-write" is being done by a prior thread. If so, the block write of the present thread is added to the partial write wait queue, as described above with reference to FIG. 13.

6. The write thread preallocates the metadata blocks.

7. The write thread releases the allocation mutex.

8. The write threads determine the state of the block write. The block write can be in one of three states, namely:

1. Partial, in-progress writes. These are writes to blocks that are on the conflict list. This write is deferred. The information to write out these blocks is added to the partial write wait queue.

2. Whole Block Writes.

3. Partial, not-in-progress writes. These are partial writes to newly allocated blocks, and are the first write to these blocks.

9. The I/O list is split apart if there are any non-contiguous areas to be written.

10. Asynchronous write requests are issued for blocks in state 2 (full block writes).

11. Synchronous read requests are issued for blocks in state 3 (Partial not-in-progress writes).

12. Asynchronous write requests are issued for blocks in state 3.

13. The write thread waits for all writes to complete, including the ones in state 1. The write thread waits for all asynchronous write callbacks. The asynchronous writes for blocks in state 1 are actually issued by other threads.

14. The write thread gets the allocation mutex.

15. The write thread commits the preallocated metadata. The allocation lists being committed are gathered together if a previous commit is in progress, and are written out under the same logging lock as described above with reference to FIG. 8.

16. The write thread removes any blocks that the write thread had added to partial block conflict queue, and issues asynchronous writes for corresponding blocks on the partial write wait queue.

17. The write thread releases the allocation mutex. The write thread has completed the write operation.

In view of the above, there have been described a multi-threaded write interface for increasing the single file write throughput of a file server. The write interface allows multiple concurrent writes to the same file and handles metadata updates using sector level locking. The write interface provides permission management to access the data blocks of the file in parallel, ensures correct use and update of indirect blocks in the tree of the file, preallocates file blocks when the file is extended, and solves access conflicts for simultaneous writes to the same block. The write interface preallocates file metadata to prevent multiple writers from allocating the same block. For example, a write operation includes obtaining a per file allocation mutex (mutually exclusive lock), reserving a metadata block, releasing the allocation mutex, issuing an asynchronous write request for writing to the file, waiting for the asynchronous write request to complete, obtaining the allocation mutex, committing the preallocated metadata block, and releasing the allocation mutex. Since no locks are held during the writing of data to the

on-disk storage and this data write takes the majority of the time, the method enhances concurrency while maintaining data integrity.

What is claimed is:

1. A method of operating a network file server for providing clients with concurrent write access to a file, the method comprising the network file server responding to a concurrent write request from a client by:

(a) obtaining a lock for the file; and then

(b) preallocating a metadata block for the file; and then

(c) releasing the lock for the file; and then

(d) asynchronously writing to the file; and then

(e) obtaining the lock for the file; and then

(f) committing the metadata block to the file; and then

(g) releasing the lock for the file.

2. The method as claimed in claim 1, wherein the file includes a hierarchy of blocks including an inode block of metadata, direct blocks of file data, and indirect blocks of metadata, and wherein the metadata block for the file is an indirect block of metadata.

3. The method as claimed in claim 2, which includes copying data from an original indirect block of the file to the metadata block for the file, the original indirect block of the file having been shared between the file and a read-only version of the file.

4. The method as claimed in claim 1, which includes concurrent writing for more than one client to the metadata block for the file.

5. The method as claimed in claim 2, wherein the asynchronous writing to the file includes a partial write to a new block that has been copied at least in part from an original block of the file, and wherein the method includes checking a partial block conflict queue for a conflict with a concurrent write to the new block, and upon failing to find an indication of a conflict with a concurrent write to the new block, preallocating the new block, copying at least a portion of the original block of the file to the new block, and performing the partial write to the new block.

6. The method as claimed in claim 2, wherein the asynchronous writing to the file includes a partial write to a new block that has been copied at least in part from an original block of the file, and wherein the method includes checking a partial block conflict queue for a conflict with a concurrent write to the new block, and upon finding an indication of a conflict with a concurrent write to the new block, waiting until resolution of the conflict with the concurrent write to the new block, and then performing the partial write to the new block.

7. The method as claimed in claim 6, which includes placing a request for the partial write in a partial write wait queue upon finding an indication of a conflict with a concurrent write to the new block, and performing the partial write upon servicing the partial write wait queue.

8. The method as claimed in claim 1, wherein the asynchronously writing to the file includes checking an input-output list for a conflicting prior concurrent access to the file, and upon finding a conflicting prior concurrent access to the file, suspending the asynchronous writing to the file until the conflicting prior concurrent access to the file is no longer conflicting.

9. The method as claimed in claim 8, wherein the suspending of the asynchronous writing to the file until the conflicting prior concurrent access is no longer conflicting provides a sector-level granularity of byte range locking for concurrent write access to the file.

10. The method as claimed in claim 1, wherein the metadata block for the file is committed by writing the metadata block to a log in storage of the network file server.

11. The method as claimed in claim 1, which includes gathering together preallocated metadata blocks for a plurality of client write requests to the file, and committing together the preallocated metadata blocks for the plurality of client write requests to the file by obtaining the lock for the file, committing the gathered preallocated metadata blocks for the plurality of client write requests to the file, and then releasing the lock for the file.

12. The method as claimed in claim 1, which includes checking whether a previous commit is in progress after asynchronously writing to the file and before obtaining the

lock for the file for committing the metadata block to the file, and upon finding that a previous commit is in progress, placing a request for committing the metadata block to the file on a staging queue for the file.

13. A method of operating a network file server for providing clients with concurrent write access to a file, the method comprising the network file server responding to a concurrent write request from a client by:

(a) preallocating a block for the file; and then

(b) asynchronously writing to the file; and then

(c) committing the block to the file;

wherein the asynchronous writing to the file includes a partial write to a new block that has been copied at least in part from an original block of the file, and wherein the method includes checking a partial block conflict queue for a conflict with a concurrent write to the new block, and upon finding an indication of a conflict with a concurrent write to the new block, waiting until resolution of the conflict with the concurrent write to the new block, and then performing the partial write to the new block.

14. The method as claimed in claim 13, wherein the method includes placing a request for the partial write in a partial write wait queue upon finding an indication of a conflict with a concurrent write to the new block, and performing the partial write upon servicing the partial write wait queue.

15. A method of operating a network file server for providing clients with concurrent write access to a file, the method comprising the network file server responding to a concurrent write request from a client by:

(a) preallocating a metadata block for the file; and then

(b) asynchronously writing to the file; and then

(c) committing the metadata block to the file;

wherein the method includes gathering together preallocated metadata blocks for a plurality of client write requests to the file, and committing together the preallocated metadata blocks for the plurality of client write requests to the file by obtaining a lock for the file, committing the gathered preallocated metadata blocks for the plurality of client write requests to the file, and then releasing the lock for the file.

16. The method as claimed in claim 15, which includes checking whether a previous commit is in progress after asynchronously writing to the file and before obtaining the lock for the file for committing the block to the file, and upon finding that a previous commit is in progress, placing a request for committing the metadata block to the file on a staging queue for the file.

17. A method of operating a network file server for providing clients with concurrent write access to a file, the method comprising the network file server responding to a concurrent write request from a client by executing a write thread, execution of the write thread including:

(a) obtaining an allocation mutex for the file; and then

(b) preallocating new metadata blocks that need to be allocated for writing to the file; and then

(c) releasing the allocation mutex for the file; and then

(d) issuing asynchronous write requests for writing to the file;

(e) waiting for callbacks indicating completion of the asynchronous write requests; and then

(f) obtaining the allocation mutex for the file; and then

(g) committing the preallocated metadata blocks; and then

(h) releasing the allocation mutex for the file.

18. A network file server comprising storage for storing a file, and at least one processor coupled to the storage for providing clients with concurrent write access to the file, wherein the network file server is programmed for responding to a concurrent write request from a client by:

(a) obtaining a lock for the file; and then

(b) preallocating a metadata block for the file; and then

(c) releasing the lock for the file; and then

(d) asynchronously writing to the file; and then

(e) obtaining the lock for the file; and then

(f) committing the metadata block to the file; and then

(g) releasing the lock for the file.

19. The network file server as claimed in claim 18, wherein the file includes a hierarchy of blocks including an inode block of metadata, direct blocks of file data, and indirect blocks of metadata, and wherein the metadata block for the file is an indirect block of metadata.

REDACTED

20. The network file server as claimed in claim 19, which is programmed for copying data from an original indirect block of the file to the metadata block for the file, the original indirect block of the file having been shared between the file and a read-only version of the file.

21. The network file server as claimed in claim 18, which is programmed for concurrent writing for more than one client to the metadata block for the file.

22. The network file server as claimed in claim 18, which includes a partial block conflict queue for indicating a concurrent write to a new block that is being copied at least in part from an original block of the file, and wherein the network file server is programmed to respond to a client request for a partial write to the new block by checking the partial block conflict queue for a conflict, and upon failing to find an indication of a conflict, preallocating the new block, copying at least a portion of the original block of the file to the new block, and performing a partial write to the new block.

23. The network file server as claimed in claim 18, which includes a partial block conflict queue for indicating a concurrent write to a new block that is being copied at least in part from an original block of the file, and wherein the network file server is programmed to respond to a client request for a partial write to the new block by checking the partial block conflict queue for a conflict, and upon finding an indication of a conflict, waiting until resolution of the conflict with the concurrent write to the new block, and then performing the partial write to the new block.

24. The network file server as claimed in claim 23, which includes a partial write wait queue, and wherein the network file server is programmed for placing a request for the

partial write in the partial write wait queue upon finding an indication of a conflict, and performing the partial write upon servicing the partial write wait queue.

25. The network file server as claimed in claim 18, which is programmed for maintaining an input-output list of concurrent reads and writes to the file, and when asynchronously writing to the file, for checking the input-output list for a conflicting prior concurrent access, read or write, to the file, and upon finding a conflicting prior concurrent access to the file, suspending the asynchronous writing to the file until the conflicting prior concurrent access to the file is no longer conflicting. The block could be read or write but must specify read explicitly and the claim should cover read access by removing the file lock (mutex) and change it to a range lock.

26. The network file server as claimed in claim 25, wherein the suspending of the asynchronous writing to the file until the conflicting prior concurrent access is no longer conflicting provides a sector-level granularity of byte range locking for concurrent write access to the file, as well as concurrent reads and writes. [Probably would be better to add a new claim.]

27. The network file server as claimed in claim 18, which is programmed for committing the metadata block for the file by writing the metadata block to a log in the storage.

28. The network file server as claimed in claim 18, which is programmed for gathering together preallocated metadata blocks for a plurality of client requests for write access to the file, and committing together the preallocated metadata blocks for the plurality of client requests for access to the file by obtaining the lock for the file, committing the gathered preallocated metadata blocks for the plurality of client requests for write access to the file, and then releasing the lock for the file.

29. The network file server as claimed in claim 18, which includes a staging queue for the file, and which is programmed for checking whether a previous commit is in progress after asynchronously writing to the file and before obtaining the lock for the file for committing the metadata block to the file, and upon finding that a previous commit is in progress, placing a request for committing the metadata block to the file on the staging queue for the file.

30. A network file server comprising storage for storing a file, and at least one processor coupled to the storage for providing clients with concurrent write access to the file, wherein the network file server is programmed for responding to a concurrent write request from a client by:

(a) preallocating a block for the file; and then

(b) asynchronously writing to the file; and then

(c) committing the block to the file;

wherein the network file server includes a partial block conflict queue for indicating a concurrent write to a new block that is being copied at least in part from an original block

of the file, and wherein the network file server is programmed for responding to a client request for a partial write to the new block by checking the partial block conflict queue for a conflict, and upon finding an indication of a conflict, waiting until resolution of the conflict with the concurrent write to the new block of the file, and then performing the partial write to the new block of the file.

31. The network file server as claimed in claim 30, which includes a partial write wait queue, and wherein the network file server is programmed for placing a request for the partial write in the partial write wait queue upon finding an indication of a conflict, and performing the partial write upon servicing the partial write wait queue.

32. A network file server comprising storage for storing a file, and at least one processor coupled to the storage for providing clients with concurrent write access to the file, wherein the network file server is programmed for responding to a concurrent write request from a client by:

(a) preallocating a metadata block for the file; and then

(b) asynchronously writing to the file; and then

(c) committing the metadata block to the file;

wherein the network file server is programmed for gathering together preallocated metadata blocks for a plurality of client write requests to the file, and committing together the preallocated metadata blocks for the plurality of client write requests to the file by obtaining a lock for the file, committing the gathered preallocated metadata blocks for the plurality of client write requests to the file, and then releasing the lock for the file.

33. The network file server as claimed in claim 32, which is programmed for checking whether a previous commit is in progress after asynchronously writing to the file and before obtaining the lock for the file for committing the metadata block to the file, and upon finding that a previous commit is in progress, placing a request for committing the metadata block to the file on a staging queue for the file.

34. A network file server comprising storage for storing a file, and at least one processor coupled to the storage for providing clients with concurrent write access to the file, wherein the network file server is programmed with a write thread for responding to a concurrent write request from a client by:

(a) obtaining an allocation mutex for the file; and then

(b) preallocating new metadata blocks that need to be allocated for writing to the file; and then

(c) releasing the allocation mutex for the file; and then

(d) issuing asynchronous write requests for writing to the file;

(e) waiting for callbacks indicating completion of the asynchronous write requests; and then

(f) obtaining the allocation mutex for the file; and then

(g) committing the preallocated metadata blocks; and then

(h) releasing the allocation mutex for the file.

35. The network file server as claimed in claim 34, which includes an uncached write interface, a file system cache and a cached read-write interface, and wherein the uncached write interface bypasses the file system cache for sector-aligned write operations.

36. The network file server as claimed in claim 35, wherein the network file server is programmed to invalidate cache blocks in the file system cache including sectors being written to by the cached read-write interface.

37. A network file server comprising storage for storing a file, and at least one processor coupled to the storage for providing clients with concurrent write access to the file, wherein the network file server is programmed for responding to a concurrent write request from a client by:

(a) preallocating a block for writing to the file;

(b) asynchronously writing to the file; and then

(c) committing the preallocated block;

wherein the network file server also includes an uncached write interface, a file system cache, and a cached read-write interface, wherein the uncached write interface bypasses the file system cache for sector-aligned write operations, and the network file server is programmed to invalidate cache blocks in the file system cache including sectors being written to by the cached read-write interface.

[Please add a new set of claims that refer explicitly to files that have read-only copies.]

ABSTRACT

A write interface in a file server provides permission management for concurrent access to data blocks of a file, ensures correct use and update of indirect blocks in a tree of the file, preallocates file blocks when the file is extended, and solves access conflicts for simultaneous writes to the same block For example, a write operation includes obtaining a per file allocation mutex (mutually exclusive lock), preallocating a metadata block, releasing the allocation mutex, issuing an asynchronous write request for writing to the file, waiting for the asynchronous write request to complete, obtaining the allocation mutex, committing the preallocated metadata block, and releasing the allocation mutex. Since no locks are held during the writing of data to the on-disk storage and this data write takes the majority of the time, the method enhances concurrency while maintaining data integrity. [Please add a sentence mentioning that the read are also faster as they do not have to wait for the file lock during a read. Also mention tha this is critical for database applications.]

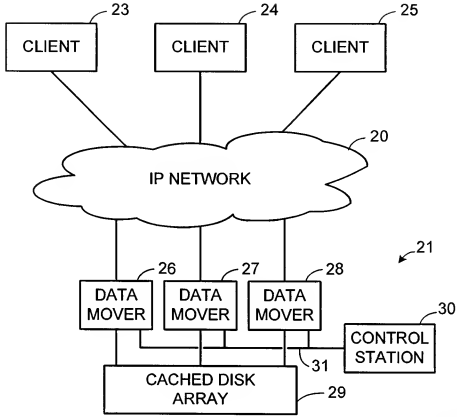
23
24
25
CLIENT
CLIENT
CLIENT
20
IP NETWORK
21
26
27
28
DATA MOVER
DATA MOVER
DATA MOVER
30
CONTROL STATION
31
CACHED DISK ARRAY
29

FIG. 1

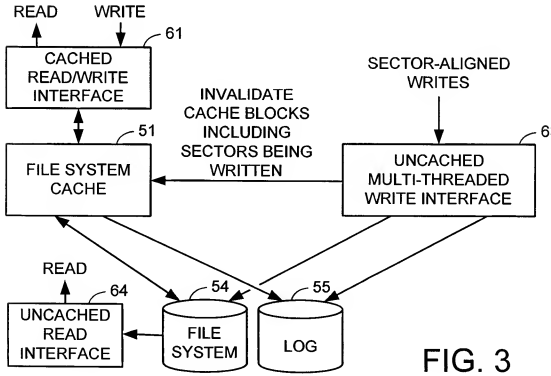
READ
WRITE
61
CACHED READ/WRITE INTERFACE
SECTOR-ALIGNED WRITES
INVALIDATE CACHE BLOCKS INCLUDING SECTORS BEING WRITTEN
51
63
FILE SYSTEM CACHE
UNCACHED MULTI-THREADED WRITE INTERFACE
READ
64
54
55
UNCACHED READ INTERFACE
FILE SYSTEM
LOG

FIG. 3

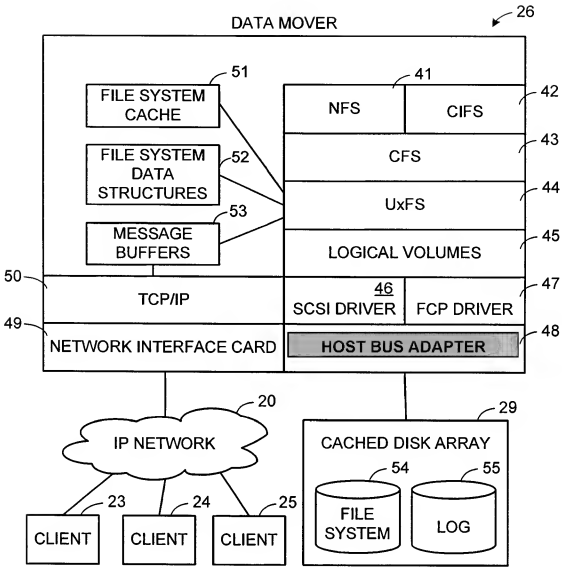
DATA MOVER
26
51
FILE SYSTEM CACHE
52
FILE SYSTEM DATA STRUCTURES
53
MESSAGE BUFFERS
41
NFS
CIFS
42
CFS
43
UxFS
44
LOGICAL VOLUMES
45
50
TCP/IP
46
SCSI DRIVER
FCP DRIVER
47
49
NETWORK INTERFACE CARD
HOST BUS ADAPTER
48
20
IP NETWORK
29
CACHED DISK ARRAY
54
55
FILE SYSTEM
LOG
23
24
25
CLIENT
CLIENT
CLIENT

FIG. 2

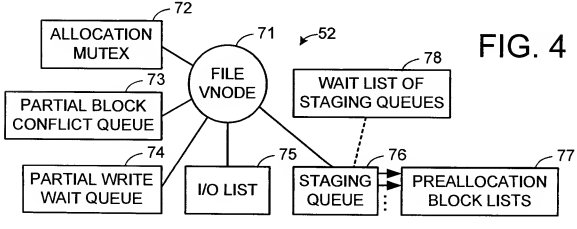
72
ALLOCATION MUTEX
71
FILE VNODE
52
78
WAIT LIST OF STAGING QUEUES
73
PARTIAL BLOCK CONFLICT QUEUE
74
PARTIAL WRITE WAIT QUEUE
75
I/O LIST
76
STAGING QUEUE
77
PREALLOCATION BLOCK LISTS

FIG. 4

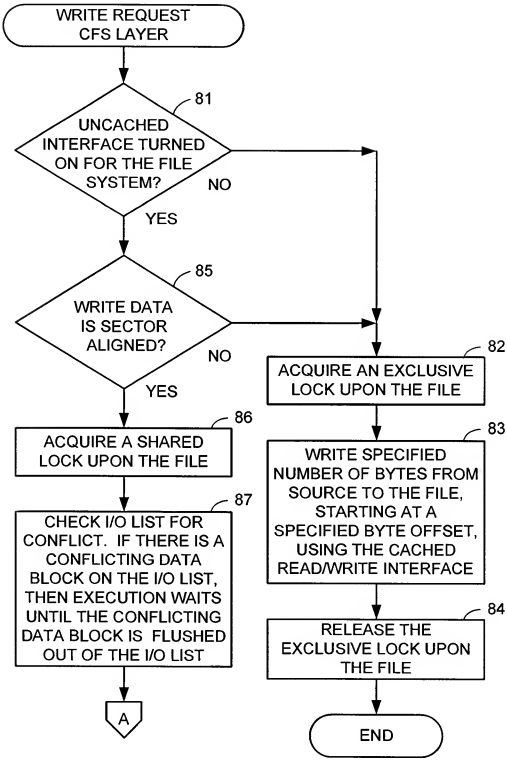
WRITE REQUEST CFS LAYER
81
UNCACHED INTERFACE TURNED ON FOR THE FILE SYSTEM?
NO
YES
85
WRITE DATA IS SECTOR ALIGNED?
NO
YES
82
ACQUIRE AN EXCLUSIVE LOCK UPON THE FILE
86
ACQUIRE A SHARED LOCK UPON THE FILE
83
WRITE SPECIFIED NUMBER OF BYTES FROM SOURCE TO THE FILE, STARTING AT A SPECIFIED BYTE OFFSET, USING THE CACHED READ/WRITE INTERFACE
87
CHECK I/O LIST FOR CONFLICT. IF THERE IS A CONFLICTING DATA BLOCK ON THE I/O LIST, THEN EXECUTION WAITS UNTIL THE CONFLICTING DATA BLOCK IS FLUSHED OUT OF THE I/O LIST
84
RELEASE THE EXCLUSIVE LOCK UPON THE FILE
A
END

FIG. 5

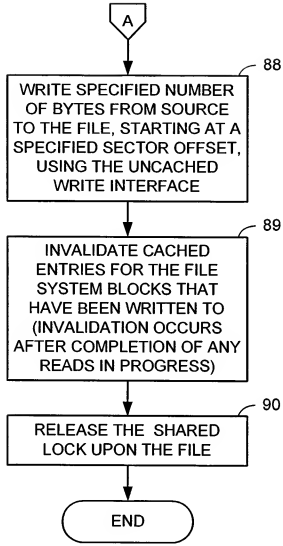
A
88
WRITE SPECIFIED NUMBER OF BYTES FROM SOURCE TO THE FILE, STARTING AT A SPECIFIED SECTOR OFFSET, USING THE UNCACHED WRITE INTERFACE
89
INVALIDATE CACHED ENTRIES FOR THE FILE SYSTEM BLOCKS THAT HAVE BEEN WRITTEN TO (INVALIDATION OCCURS AFTER COMPLETION OF ANY READS IN PROGRESS)
90
RELEASE THE SHARED LOCK UPON THE FILE
END

FIG. 6

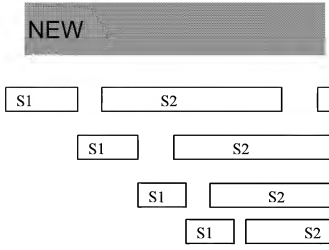

Figure 50 Write Request Pipelining: Writes are divided into three stages

S1: preallocation of metadata
S2: data writes
S3: Commit of metadata

Fig. 50

NEW

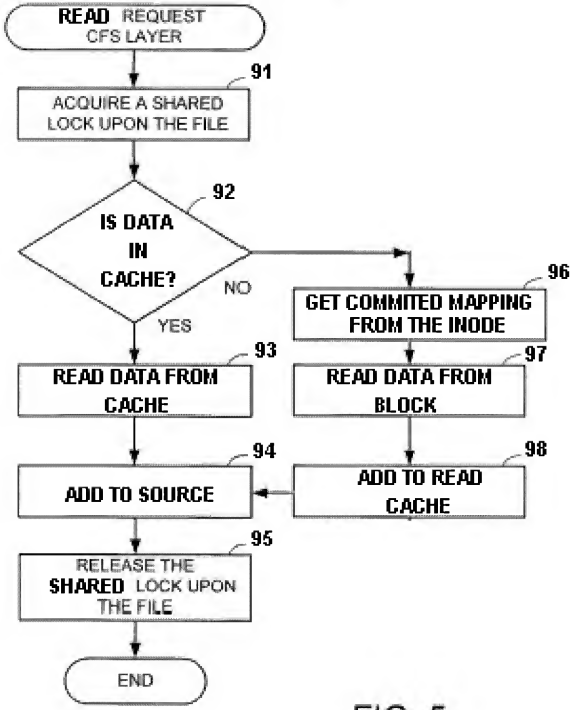
READ REQUEST CFS LAYER
91
ACQUIRE A SHARED LOCK UPON THE FILE
92
IS DATA IN CACHE?
NO
YES
96
GET COMMITED MAPPING FROM THE INODE
93
READ DATA FROM CACHE
97
READ DATA FROM BLOCK
94
ADD TO SOURCE
98
ADD TO READ CACHE
95
RELEASE THE SHARED LOCK UPON THE FILE
END

FIG. 5a

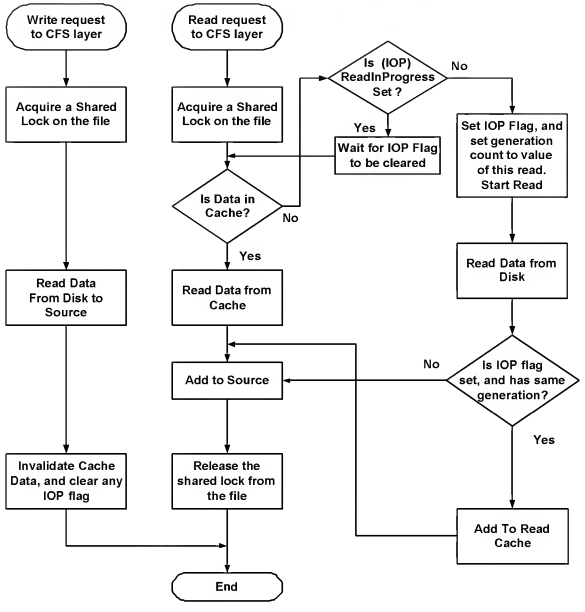
Write request to CFS layer
Read request to CFS layer
Is (IOP) ReadInProgress Set ?
No
Yes
Acquire a Shared Lock on the file
Acquire a Shared Lock on the file
Wait for IOP Flag to be cleared
Set IOP Flag, and set generation count to value of this read. Start Read
Is Data in Cache?
No
Yes
Read Data from Disk
Read Data From Disk to Source
Read Data from Cache
No
Is IOP flag set, and has same generation?
Add to Source
Yes
Invalidate Cache Data, and clear any IOP flag
Release the shared lock from the file
Add To Read Cache
End

Fig 5b

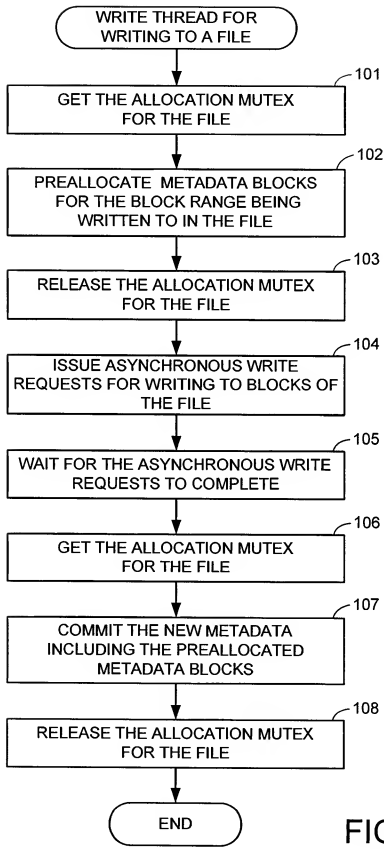
WRITE THREAD FOR WRITING TO A FILE
101
GET THE ALLOCATION MUTEX FOR THE FILE
102
PREALLOCATE METADATA BLOCKS FOR THE BLOCK RANGE BEING WRITTEN TO IN THE FILE
103
RELEASE THE ALLOCATION MUTEX FOR THE FILE
104
ISSUE ASYNCHRONOUS WRITE REQUESTS FOR WRITING TO BLOCKS OF THE FILE
105
WAIT FOR THE ASYNCHRONOUS WRITE REQUESTS TO COMPLETE
106
GET THE ALLOCATION MUTEX FOR THE FILE
107
COMMIT THE NEW METADATA INCLUDING THE PREALLOCATED METADATA BLOCKS
108
RELEASE THE ALLOCATION MUTEX FOR THE FILE
END

FIG. 7

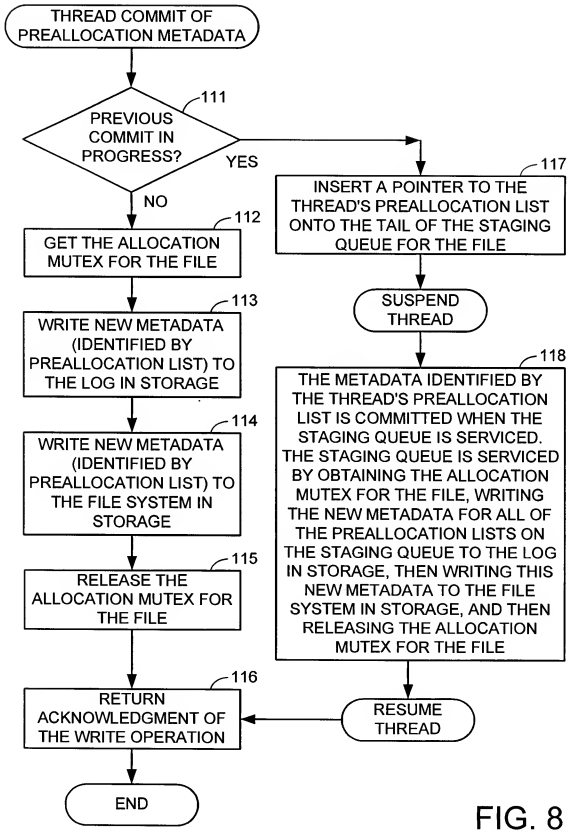
THREAD COMMIT OF PREALLOCATION METADATA
111
PREVIOUS COMMIT IN PROGRESS?
YES
NO
117
INSERT A POINTER TO THE THREAD'S PREALLOCATION LIST ONTO THE TAIL OF THE STAGING QUEUE FOR THE FILE
SUSPEND THREAD
112
GET THE ALLOCATION MUTEX FOR THE FILE
113
WRITE NEW METADATA (IDENTIFIED BY PREALLOCATION LIST) TO THE LOG IN STORAGE
114
WRITE NEW METADATA (IDENTIFIED BY PREALLOCATION LIST) TO THE FILE SYSTEM IN STORAGE
115
RELEASE THE ALLOCATION MUTEX FOR THE FILE
118
THE METADATA IDENTIFIED BY THE THREAD'S PREALLOCATION LIST IS COMMITTED WHEN THE STAGING QUEUE IS SERVICED. THE STAGING QUEUE IS SERVICED BY OBTAINING THE ALLOCATION MUTEX FOR THE FILE, WRITING THE NEW METADATA FOR ALL OF THE PREALLOCATION LISTS ON THE STAGING QUEUE TO THE LOG IN STORAGE, THEN WRITING THIS NEW METADATA TO THE FILE SYSTEM IN STORAGE, AND THEN RELEASING THE ALLOCATION MUTEX FOR THE FILE
116
RETURN ACKNOWLEDGMENT OF THE WRITE OPERATION
RESUME THREAD
END

FIG. 8

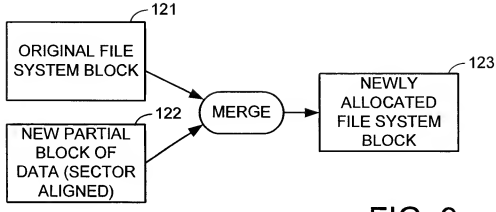
121
ORIGINAL FILE SYSTEM BLOCK
122
NEW PARTIAL BLOCK OF DATA (SECTOR ALIGNED)
MERGE
123
NEWLY ALLOCATED FILE SYSTEM BLOCK

FIG. 9

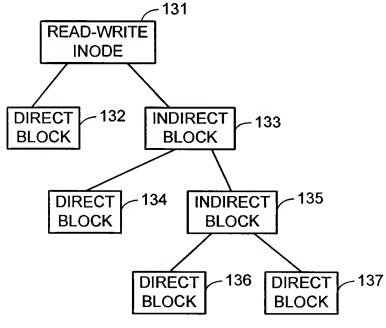
131
READ-WRITE INODE
DIRECT BLOCK
132
INDIRECT BLOCK
133
DIRECT BLOCK
134
INDIRECT BLOCK
135
DIRECT BLOCK
136
DIRECT BLOCK
137

FIG. 10

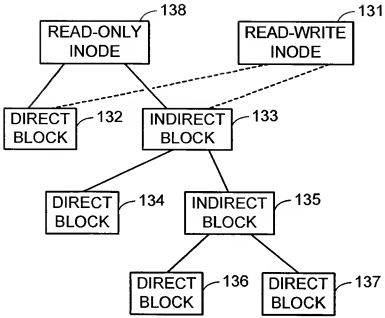
138
READ-ONLY INODE
131
READ-WRITE INODE
DIRECT BLOCK
132
INDIRECT BLOCK
133
DIRECT BLOCK
134
INDIRECT BLOCK
135
DIRECT BLOCK
136
DIRECT BLOCK
137

FIG. 11

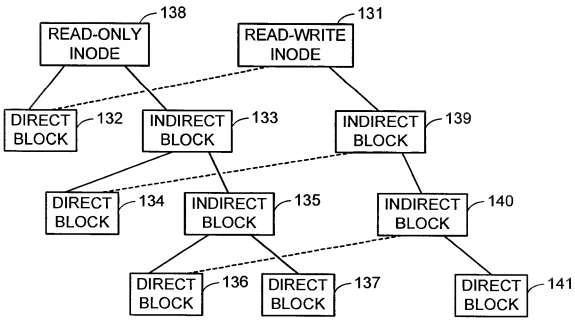
138
READ-ONLY INODE
131
READ-WRITE INODE
DIRECT BLOCK
132
INDIRECT BLOCK
133
INDIRECT BLOCK
139
DIRECT BLOCK
134
INDIRECT BLOCK
135
INDIRECT BLOCK
140
DIRECT BLOCK
136
DIRECT BLOCK
137
DIRECT BLOCK
141

FIG. 12

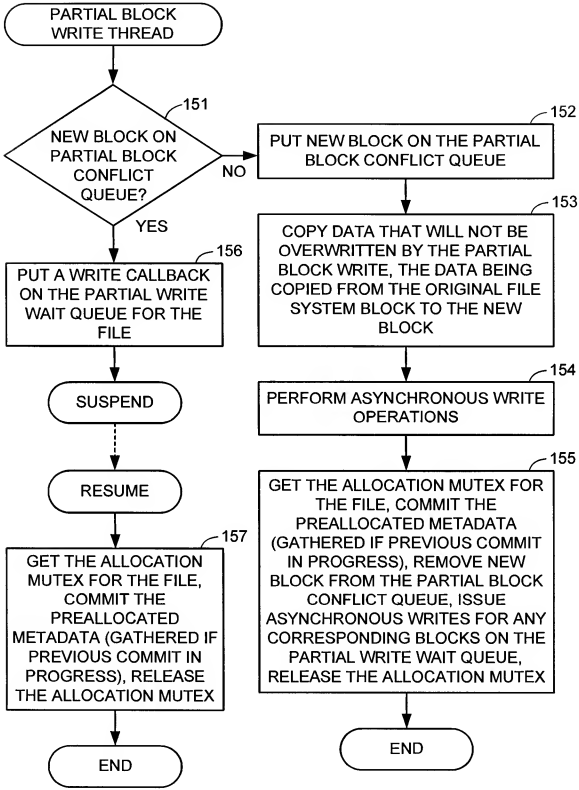
PARTIAL BLOCK WRITE THREAD
151
NEW BLOCK ON PARTIAL BLOCK CONFLICT QUEUE?
NO
YES
152
PUT NEW BLOCK ON THE PARTIAL BLOCK CONFLICT QUEUE
153
COPY DATA THAT WILL NOT BE OVERWRITTEN BY THE PARTIAL BLOCK WRITE, THE DATA BEING COPIED FROM THE ORIGINAL FILE SYSTEM BLOCK TO THE NEW BLOCK
154
PERFORM ASYNCHRONOUS WRITE OPERATIONS
155
GET THE ALLOCATION MUTEX FOR THE FILE, COMMIT THE PREALLOCATED METADATA (GATHERED IF PREVIOUS COMMIT IN PROGRESS), REMOVE NEW BLOCK FROM THE PARTIAL BLOCK CONFLICT QUEUE, ISSUE ASYNCHRONOUS WRITES FOR ANY CORRESPONDING BLOCKS ON THE PARTIAL WRITE WAIT QUEUE, RELEASE THE ALLOCATION MUTEX
END
156
PUT A WRITE CALLBACK ON THE PARTIAL WRITE WAIT QUEUE FOR THE FILE
SUSPEND
RESUME
157
GET THE ALLOCATION MUTEX FOR THE FILE, COMMIT THE PREALLOCATED METADATA (GATHERED IF PREVIOUS COMMIT IN PROGRESS), RELEASE THE ALLOCATION MUTEX
END

FIG. 13

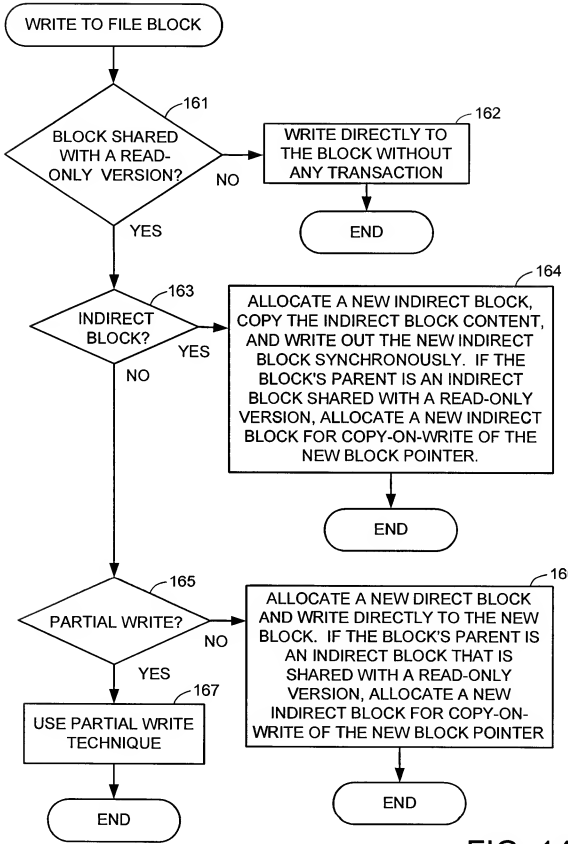
WRITE TO FILE BLOCK
161
BLOCK SHARED WITH A READ-ONLY VERSION?
NO
YES
162
WRITE DIRECTLY TO THE BLOCK WITHOUT ANY TRANSACTION
END
163
INDIRECT BLOCK?
YES
NO
164
ALLOCATE A NEW INDIRECT BLOCK, COPY THE INDIRECT BLOCK CONTENT, AND WRITE OUT THE NEW INDIRECT BLOCK SYNCHRONOUSLY. IF THE BLOCK'S PARENT IS AN INDIRECT BLOCK SHARED WITH A READ-ONLY VERSION, ALLOCATE A NEW INDIRECT BLOCK FOR COPY-ON-WRITE OF THE NEW BLOCK POINTER.
END
165
PARTIAL WRITE?
NO
YES
166
ALLOCATE A NEW DIRECT BLOCK AND WRITE DIRECTLY TO THE NEW BLOCK. IF THE BLOCK'S PARENT IS AN INDIRECT BLOCK THAT IS SHARED WITH A READ-ONLY VERSION, ALLOCATE A NEW INDIRECT BLOCK FOR COPY-ON-WRITE OF THE NEW BLOCK POINTER
END
167
USE PARTIAL WRITE TECHNIQUE
END

FIG. 14

EXHIBIT C

Auchterlonie, Richard

From: Auchterlonie, Richard
Sent: Thursday, September 04, 2003 12:59 PM
To: 'sfaibish@emc.com'
Cc: 'Clark_William@emc.com'; 'Mazzarella_Julie@emc.com'
Subject: EMC-03-062 A New Method for Enabling Simultaneous Parallel Writes to a Single File

Sorin:

Please find attached a second draft of the specification, drawings, declaration, and assignment for your patent application. Please review and pass it along to the other inventors as appropriate. If you would like any changes, please let me know and I will send you a revised draft. Otherwise, once all of the inventors have approved of the patent application, please contact Julie Mazzarella to arrange for execution of the patent application.

Thanks,

Richard C. Auchterlonie
Howrey Simon Arnold & White, LLP
750 Bering Drive
Houston, Texas 77057
Phone: (713) 787-1698
Fax: (713) 787-1440
AuchterlonieR@howrey.com


EMCR 100 Drawings.pdf


EMCR 100 PA Parallel Writes.do...


EMCR100 Assignment.DOC


EMCR100 Declaration.DOC